C Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

May 1, 1977

1. Introduction

C is a computer language which offers a rich selection of operators and data types and the ability to impose useful structure on both control flow and data. All the basic operations and data objects are close to those actually implemented by most real computers, so that a very efficient implementation is possible, but the design is not tied to any particular machine and with a little care it is possible to write easily portable programs.

This manual describes the current version of the C language as it exists on the PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

Blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. Some space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore '_' counts as alphabetic. Upper and lower case letters are considered different. On the PDP-11, no more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

The **entry** keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the word **fortran**.

2.4 Constants

There are several kinds of constants, as follows:

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero), decimal otherwise. The digits **8** and **9** have octal value 10 and 11 respectively. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer (32767 on the PDP-11) is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer (0177777 or 0xFFFF on the PDP-11) is likewise taken to be **long**.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant, which, on the PDP-11, has 32 significant bits. As discussed below, on other machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a sequence of characters enclosed in single quotes ' '. Within a character constant a single quote must be preceded by a backslash '\'. Certain non-graphic characters, and '\' itself, may be escaped according to the following table:

BS	\b
NL (LF)	\n
CR	\r
HT	\t
FF	\f
ddd	\ddd
\	\\\\

The escape '\ddd' consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is '\0' (not followed by a digit) which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash vanishes.

The value of a single-character constant is the numerical value of the character in the machine's character set (ASCII for the PDP-11). On the PDP-11 at most two characters are permitted in a character constant and the second character of a pair is stored in the high-order byte of the integer value. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes ' " '. A string has type 'array of characters' and storage class 'static' (see below) and is initialized with the given characters. The compiler places a null byte '\0' at the end of each string so that programs which scan the string can find its end. In a string, the character ' " ' must be preceded by a '\'; in addition, the same escapes as described for character constants may be used. Finally, a '\' and an immediately following new-line are ignored.

All strings, even when written identically, are distinct.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **sans-serif** type. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript 'opt,' so that

$\{ \; expression_{opt} \; \}$

would indicate an optional expression in braces. The complete syntax is given in §16, in the notation of YACC.

4. What's in a Name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block, and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and if a genuine character is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent. On the PDP-11, characters are stored as signed 8-bit integers, and the character set is ASCII.

Up to three sizes of integer, declared **short int**, **int**, and **long int** are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both equivalent to plain integers. 'Plain' integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs. On the PDP-11, short and plain integers are both represented in 16-bit 2's complement notation. Long integers are 32-bit 2's complement.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (16 on the PDP-11; long and short unsigned quantities are not supported.)

Single precision floating point (**float**) quantities, on the PDP-11, have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (**double**) quantities on the PDP-11 have the same range as **floats** and a precision of 56 bits or about 17 decimal digits. Some implementations may make **float** and **double** synonymous.

Because objects of these types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will collectively be called *integral* types. **Float** and **double** will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing a sequence of objects of various types;

unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name 'lvalue' comes from the assignment expression '$E1 = E2$' in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a short integer always involves sign extension; short integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. On the PDP-11, character variables range in value from −128 to 127; a character constant specified using an octal escape also suffers sign extension and may appear negative, for example ''\214''.

When a longer integer is converted to a shorter or to a **char**, it is truncated on the left.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a **float** appears in an expression it is lengthened to **double** by zero-padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent. On the PDP-11, truncation is towards 0. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value (on the PDP-11) is the least unsigned integer congruent to the signed integer (modulo 2^{16}). Because of the 2's complement notation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the 'usual arithmetic conversions.'

First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.

Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.

Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.

Otherwise, both operands must be **int**, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the collected grammar.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. Expressions involving a commutative and associative operator may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

7.1 Primary expressions

Primary expressions involving ., −>, subscripting, and function calls group left to right.

> *primary-expression:*
> > *identifier*
> > *constant*
> > *string*
> > (*expression*)
> > *primary-expression* [*expression*]
> > *primary-expression* (*expression-list*$_{opt}$)
> > *primary-lvalue* . *identifier*
> > *primary-expression* −> *identifier*
>
> *expression-list:*
> > *expression*
> > *expression-list* , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is 'array of ...', then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is 'pointer to ...'. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared 'function returning ...', when used except in the function-name position of a call, is converted to 'pointer to function returning ...'.

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form.

A string is a primary expression. Its type is originally 'array of **char**'; but following the same rule given above for identifiers, this is modified to 'pointer to **char**' and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type 'pointer to ...', the subscript expression is **int**, and the type of the result is '...'. The expression 'E1[E2]' is identical (by definition) to '*((E1)+(E2))'. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type 'function returning ...', and the result of the function call is of type '...'. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call; any of type **char** or **short** are converted to **int**.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a '−' and a '>') followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression 'E1−>MOS' is the same as '(*E1).MOS'. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
- * *expression*
- & *lvalue*
- − *expression*
- ! *expression*
- ~ *expression*
- ++ *lvalue*
- −− *lvalue*
- *lvalue* ++
- *lvalue* −−
- (*type-name*) *expression*
- **sizeof** *expression*
- **sizeof** (*type-name*)

The unary * operator means *indirection:* the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is 'pointer to ...', the type of the result is '...'.

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is '...', the type of the result is 'pointer to ...'.

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is 16 on the PDP-11.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix '++' is incremented. The value is the new value of the operand, but is not an lvalue. The expression '++a' is equivalent to '(a += 1)'. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix '−−' is decremented analogously to the ++ operator.

When postfix '++' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix '−−' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix −− operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. The construction of type names is described in §8.7.

The **sizeof** operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof.** However in all existing implementations a byte is the space required to hold a **char.**) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction 'sizeof(type)' is taken to be a unit, so the expression 'sizeof(type)−2' is the same as '(sizeof(type))−2'.

7.3 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
*expression * expression*
expression / expression
expression % expression

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. In all cases it is true that $(a/b)*b + a\%b = a$. On the PDP-11, the remainder has the same sign as the dividend.

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. On the PDP-11, the remainder has the same sign as the dividend. The operands must not be floating.

7.4 Additive operators

The additive operators + and − group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + expression
expression − expression

The result of the '+' operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if P is a pointer to an object in an array, the expression 'P+1' is a pointer to the next object in the array.

No further type combinations are allowed.

The + operator is associative and expressions with several additions at the same level may be rearranged.

The result of the '−' operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or larger than the number of bits in the object.

shift-expression:
- *expression* << *expression*
- *expression* >> *expression*

The value of 'E1<<E2' is E1 (interpreted as a bit pattern) left-shifted E2 bits; vacated bits are 0-filled. The value of 'E1>>E2' is E1 right-shifted E2 bit positions. The shift is guaranteed to be logical (0-fill) if E1 is **unsigned**; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; 'a<b<c' does not mean what it seems to.

relational-expression:
- *expression* < *expression*
- *expression* > *expression*
- *expression* <= *expression*
- *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared, and the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
- *expression* == *expression*
- *expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus 'a<b == c<d' is 1 whenever a<b and c<d have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise *and* operator

and-expression:
- *expression* & *expression*

--

The **&** operator is associative and expressions involving **&** may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'and' function of the operands. The operator applies only to integral operands.

7.9 Bitwise *exclusive or* operator

exclusive-or-expression:
  expression **^** *expression*

The **^** operator is associative and expressions involving **^** may be rearranged. The usual arithmetic conversions are performed; the result is is the bit-wise 'exclusive or' function of the operands. The operator applies only to integral operands.

7.10 Bitwise *inclusive or* operator

inclusive-or-expression:
  expression **|** *expression*

The **|** operator is associative and expressions with **|** may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'inclusive or' function of its operands. The operator applies only to integral operands.

7.11 Logical *and* operator

logical-and-expression:
  expression **&&** *expression*

The **&&** operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike **&**, **&&** guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.12 Logical *or* operator

logical-or-expression:
  expression **||** *expression*

The **||** operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike **|**, **||** guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.13 Conditional operator

conditional-expression:
  expression **?** *expression* **:** *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
    lvalue = *expression*
    lvalue + = *expression*
    lvalue − = *expression*
    lvalue * = *expression*
    lvalue / = *expression*
    lvalue % = *expression*
    lvalue >> = *expression*
    lvalue << = *expression*
    lvalue & = *expression*
    lvalue ^ = *expression*
    lvalue | = *expression*

Notice that the representation of the compound assignment operators has changed; formerly the '=' came first and the other operator came second (without any space). The compiler continues to accept the previous notation.

In the simple assignment with '=', the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form 'E1 op = E2' may be inferred by taking it as equivalent to 'E1 = E1 op (E2)'; however, E1 is evaluated only once. In += and −=, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compiler currently allows a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
    expression , *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example, 'f(a, (t = 3, t+2), c)' has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
    decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

> *decl-specifiers:*
>> *type-specifier decl-specifiers$_{opt}$*
>> *sc-specifier decl-specifiers$_{opt}$*

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

> *sc-specifier:*
>> **auto**
>> **static**
>> **extern**
>> **register**
>> **typedef**

The **typedef** specifier does not reserve storage and is called a 'storage class specifier' only for syntactic convenience; it is discussed in §8.8.

The meanings of the various storage classes were discussed in §4.

The **auto, static,** and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few (three, for the PDP-11) such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int, char,** or pointer. One restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future developments may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are always **extern.**

8.2 Type specifiers

The type-specifiers are

> *type-specifier:*
>> **char**
>> **short**
>> **int**
>> **long**
>> **unsigned**
>> **float**
>> **double**
>> *struct-or-union-specifier*
>> *typedef-name*

The words **long, short,** and **unsigned** may be thought of as adjectives; the following combinations are acceptable (in any order).

>> **short int**
>> **long int**
>> **unsigned int**
>> **long float**

The meaning of the last is the same as **double.** Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be **int.**

Specifiers for structures and unions are discussed in §8.5; declarations with **typedef** names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
  init-declarator
  init-declarator , *declarator-list*

init-declarator:
  declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
  identifier
  (*declarator*)
  * *declarator*
  declarator ( )
  declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

If a declarator has the form

```
* D
```

for D a declarator, then the contained identifier has the type 'pointer to ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

```
D( )
```

then the contained identifier has the type 'function returning ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

```
D[constant-expression]
```

or

```
D[ ]
```

Such declarators make the contained identifier have type 'array.' If the unadorned declarator D would specify a non-array of type '...', then the declarator 'D[i]' yields a 1-dimensional array with rank i of objects of type '...'. If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n$, then the declarator D[$i_{n+1}$] yields an ($n$+1)-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n \times i_{n+1}$.

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is **int**. (Constant expressions are defined precisely in §15.) The constant expression of an array declarator may be missing only for the first dimension. This notation is useful when the array is external and the actual declaration, which allocates storage, is given elsewhere. The constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f ( ), *fip( ), (*pfi) ( );
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. It is especially useful to compare the last two. The binding of '*fip()' is '*(fip())', so that the declaration suggests, and the same construction in an expression requires, the calling of a function *fip*, and then using indirection through the (pointer) result to yield an integer. In the declarator '(*pfi) ()', the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, *x3d* is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions 'x3d', 'x3d[i]', 'x3d[i][j]', 'x3d[i][j][k]' may reasonably appear in an expression. The first three have type 'array', the last has type **int**.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

structure-or-union-specifier:
    struct-or-union { *struct-decl-list* }
    struct-or-union identifier { *struct-decl-list* }
    struct-or-union identifier

struct-or-union:
    struct
    union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
    struct-declaration
    struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , *struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field;* its length is set off from the field name by a colon.

struct-declarator:
  declarator
  declarator : *constant-expression*
  : *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. On the PDP-11, fields are assigned right-to-left.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The 'next field' presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even int fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

  struct *identifier* { *struct-decl-list* }
  union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

  struct *identifier*
  union *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure or union which contains an instance of itself, as distinct from a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp−>count
```

refers to the *count* field of the structure to which *sp* points;

```
s.left
```

refers to the left subtree pointer of the structure *s*. Finally,

```
s.right−>tword[0]
```

refers to the first character of the *tword* member of the right subtree of *s*.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by '=', and consists of an expression or a list of values nested in braces.

initializer:
  = *expression*
  = { *initializer-list* }
  = { *initializer-list* , }

initializer-list:
  expression
  initializer-list , *initializer-list*
  { *initializer-list* }

The '=' is a new addition to the syntax, intended to alleviate potential ambiguities. The current compiler allows it to be omitted when the rest of the initializer is a very simple expression (just a name, string, or constant) or when the rest of the initializer is enclosed in braces.

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving previously declared variables.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates. Currently, the PDP-11 compiler also forbids initializing fields in structures.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeding comma-separated list of initializers initialize the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive members of the string initialize the members of the array.

For example,

```
int x[ ] = { 1, 3, 5 };
```

declares and initializes *x* as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float  y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array *y*[0], namely *y*[0][0], *y*[0][1], and *y*[0][2]. Likewise the next two lines initialize *y*[1] and *y*[2]. The initializer ends early and therefore *y*[3] is initialized with 0. Precisely the same effect could have been achieved by

```
float  y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7,
};
```

The initializer for *y* begins with a left brace, but that for *y*[0] does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for *y*[1] and *y*[2]. Also,

```
float  y[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of *y* (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char  msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly, and as an argument of **sizeof**) it is desired to supply the name of a data type. This is accomplished using a 'type name,' which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
  type-specifier abstract-declarator

abstract-declarator:
    empty
    (*abstract-declarator*)
    * *abstract-declarator*
    abstract-declarator ( )
    abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

    (*abstract-declarator*)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
```

name respectively the types 'integer,' 'pointer to integer,' 'array of 3 pointers to integers,' and 'pointer to an array of 3 integers.' As another example,

```
int i;
. . .
sin( (double) i);
```

calls the *sin* routine (which accepts a **double** argument) with an argument appropriately converted.

8.8 Typedef

Declarations whose 'storage class' is **typedef** do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types. Within the scope of a declaration involving **typedef**, each of the identifiers appearing as part of any declarators therein become syntactically equivalent to type keywords naming the type associated with the identifiers in the way described in §8.4.

typedef-name:
    identifier

For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of *distance* is '**int**', that of *metricp* is 'pointer to **int**,' and that of *z* is the specified structure. *Zp* is a pointer to such a structure.

Typedef does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above *distance* is considered to have exactly the same type as any other **int** variable.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called 'block') is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, at which time it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **external** declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the 'else' ambiguity is resolved by connecting an **else** with the last encountered elseless **if**.

9.4 While statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The **for** statement has the form

for (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*

This statement is equivalent to

```
expression-1;
while (expression-2) {
        statement
        expression-3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing expression-2 makes the implied **while** clause equivalent to 'while(1)'; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labelled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a **default** prefix, control passes to the prefixed statement. If no case matches and if there is no **default** then none of the statements in the switch is executed.

Case and **default** prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see **break**, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while ( ... ) {          do {                   for ( ... ) {
  ...                      ...                    ...
contin: ;                contin: ;              contin: ;
}                        } while ( ... ) ;      }
```

a **continue** is equivalent to 'goto contin'. (Following the 'contin:' is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the **return** statement, which has one of the forms

return ;
return *expression* ;

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (§9.12) located in the current function. Previous versions of C had an incompletely implemented notion of label variable, which has been withdrawn.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the '}' of a compound statement or to supply a null body to a looping statement such as **while**.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
  decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; See §11.2 for the distinction between them. A function declarator is similar to a declarator for a 'function returning ...' except that it lists the formal parameters of the function being defined.

function-declarator:
  declarator (*parameter-list*$_{opt}$)

parameter-list:
  identifier
  identifier , *parameter-list*

The function-body has the form

function-body:
  declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
	int m;
	m = (a>b)? a:b;
	return(m>c? m:c);
}
```

Here 'int' is the type-specifier; 'max(a, b, c)' is the function-declarator; 'int a, b, c;' is the declaration-list for the formal parameters; '{ ... }' is the block giving the code for the statement. The parentheses in the **return** are not required.

C converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared 'array of ...' are adjusted to read 'pointer to ...'. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free **return** statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
  declaration

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing 'undefined identifier' diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. **Typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
. . .
{    auto int distance;
     . . .
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance.***

11.2 Scope of externals

If a function declares an identifier to be **extern**, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, compatible external definitions of the same identifier may be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. The appearance of the **extern** keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an

*It is agreed that the ice is thin here.

external data definition without the **extern** specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the **extern** in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visible in other files.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with '#' communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

define *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

define *identifier*(*identifier* , ... , *identifier*) *token-string*

where there is no space between the first identifier and the '(', is a macro definition with arguments. Subsequent instances of the first identifier followed by a '(', a sequence of tokens delimited by commas, and a ')' are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing '\' at the end of the line to be continued.

This facility is most valuable for definition of 'manifest constants', as in

```
# define TABSIZE 100
...
int table[TABSIZE];
```

A control line of the form

undef *identifier*

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

include "*filename*"

causes the replacement of that line by the entire contents of the file *filename*.

The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

include <filename>

searches only the standard places, and not the directory of the source file.

Includes may be nested.

12.3 Conditional compilation

A compiler control line of the form

> # **if** *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

> # **ifdef** *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a **#define** control line. A control line of the form

> # **ifndef** *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

> # **else**

and then by a control line

> # **endif**

If the checked condition is true then any lines between **#else** and **#endif** are ignored. If the checked condition is false then any lines between the test and an **#else** or, lacking an **#else**, the **#endif**, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

> # **line** *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the next line number is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions, since **auto** functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is 'function returning ...', it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not currently declared is contextually declared to be 'function returning **int**'.

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the . operator); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or −>) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before '.', and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a '−>' is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f( );
...
g(f);
```

Then the definition of *g* might read

```
g(funcp)
int (*funcp) ( );
{
        ...
        (*funcp) ( );
        ...
}
```

Notice that *f* was declared explicitly in the calling routine since its first appearance was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that 'E1[E2]' is identical to '*((E1) + (E2))'. Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an *n*-dimensional array of rank $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here *x* is a 3×5 array of integers. When *x* appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression 'x[i]', which is equivalent to '*(x+i)', *x* is first converted to a pointer as described; then *i* is converted to the type of *x*, which involves multiplying *i* by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and **sizeof** expressions, possibly connected by the binary operators

```
+ - * / % & | ^ << >> == != < > <= >=
```

or by the unary operators

```
- ~
```

or by the ternary operator

```
? :
```

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Grammar revisited.

This section repeats the grammar of C in notation somewhat different than given before. The description below is adapted directly from a YACC grammar actually used by several compilers; thus it may (aside from possible editing errors) be regarded as authentic. The notation is pure YACC with the exception that the '|' separating alternatives for a production is omitted, since alternatives are always on separate lines; the ';' separating productions is omitted since a blank line is left between productions.

The lines with '%term' name the terminal symbols, which are either commented upon or should be self-evident. The lines with '%left,' '%right,' and '%binary' indicate whether the listed terminals are left-associative, right-associative, or non-associative, and describe a precedence structure. The precedence (binding strength) increases as one reads down the page. When the construction '%prec *x*' appears the precedence of the rule is that of the terminal *x*; otherwise the precedence of the rule is that of its leftmost terminal.

```
%term NAME
%term STRING
%term ICON
%term FCON
%term PLUS
%term MINUS
%term MUL
%term AND
%term QUEST
%term COLON
%term ANDAND
%term OROR
%term ASOP      /* old-style =+ etc. */
%term RELOP     /* <= >= < > */
%term EQUOP     /* == != */
%term DIVOP     /* / % */
%term OR        /* | */
%term EXOR      /* ^ */
%term SHIFTOP   /* << >> */
%term INCOP     /* ++ -- */
%term UNOP      /* ! ~ */
%term STROP     /* . -> */


%term TYPE      /* int, char, long, float, double, unsigned, short */
%term CLASS     /* extern, register, auto, static, typedef */
%term STRUCT    /* struct or union */
%term RETURN
%term GOTO
%term IF
%term ELSE
%term SWITCH
%term BREAK
%term CONTINUE
%term WHILE
%term DO
%term FOR
%term DEFAULT
%term CASE
%term SIZEOF
```

```
%term LP        /* ( */
%term RP        /* ) */
%term LC        /* { */
%term RC        /* } */
%term LB        /* [ */
%term RB        /* ] */
%term CM        /* , */
%term SM        /* ; */
%term ASSIGN    /* = */


%left    CM
%right   ASOP      ASSIGN
%right   QUEST     COLON
%left    OROR
%left    ANDAND
%left    OROP
%left    AND
%binary  EQUOP
%binary  RELOP
%left    SHIFTOP
%left    PLUS      MINUS
%left    MUL       DIVOP
%right   UNOP
%right   INCOP     SIZEOF
%left    LB        LP        STROP


program:          ext_def_list

ext_def_list:     ext_def_list external_def
                  /* empty */

external_def:     optattrib SM
                  optattrib init_dcl_list SM
                  optattrib fdeclarator function_body

function_body:    dcl_list compoundstmt

dcl_list:         dcl_list declaration
                  /* empty */

declaration:      specifiers declarator_list SM
                  specifiers SM

optattrib:        specifiers
                  /* empty */

specifiers:       CLASS type
                  type CLASS
                  CLASS
                  type
```

```
type:              TYPE
                   TYPE TYPE
                   struct_dcl

struct_dcl:        STRUCT NAME LC type_dcl_list RC
                   STRUCT LC type_dcl_list RC
                   STRUCT NAME

type_dcl_list:     type_declaration
                   type_dcl_list type_declaration

type_declaration:  type declarator_list SM
                   struct_dcl SM
                   type SM

declarator_list:   declarator
                   declarator_list CM declarator

declarator:        fdeclarator
                   nfdeclarator
                   nfdeclarator COLON con_e %prec CM
                   COLON con_e %prec CM

nfdeclarator:      MUL nfdeclarator
                   nfdeclarator LP RP
                   nfdeclarator LB RB
                   nfdeclarator LB con_e RB
                   NAME
                   LP nfdeclarator RP

fdeclarator:       MUL fdeclarator
                   fdeclarator LP RP
                   fdeclarator LB RB
                   fdeclarator LB con_e RB
                   LP fdeclarator RP
                   NAME LP name_list RP
                   NAME LP RP

name_list:         NAME
                   name_list CM NAME

init_dcl_list:     init_declarator %prec CM
                   init_dcl_list CM init_declarator

init_declarator:   nfdeclarator
                   nfdeclarator ASSIGN initializer
                   nfdeclarator initializer
                   fdeclarator

init_list:         initializer %prec CM
                   init_list CM initializer

initializer:       e %prec CM
                   LC init_list RC
```

```
               LC init_list CM RC

compoundstmt:  LC dcl_list stmt_list RC

stmt_list:     stmt_list statement
               /* empty */

statement:     e SM
               compoundstmt
               IF LP e RP statement
               IF LP e RP statement ELSE statement
               WHILE LP e RP statement
               DO statement WHILE LP e RP SM
               FOR LP opt_e SM opt_e SM opt_e RP statement
               SWITCH LP e RP statement
               BREAK SM
               CONTINUE SM
               RETURN SM
               RETURN e SM
               GOTO NAME SM
               SM
               label statement

label:         NAME COLON
               CASE con_e COLON
               DEFAULT COLON

con_e:         e %prec CM

opt_e:         e
               /* empty */

elist:         e %prec CM
               elist CM e

e:             e MUL e
               e CM e
               e DIVOP e
               e PLUS e
               e MINUS e
               e SHIFTOP e
               e RELOP e
               e EQUOP e
               e AND e
               e OROP e
               e ANDAND e
               e OROR e
               e MUL ASSIGN e
               e DIVOP ASSIGN e
               e PLUS ASSIGN e
               e MINUS ASSIGN e
               e SHIFTOP ASSIGN e
               e AND ASSIGN e
               e OROP ASSIGN e
```

```
                e QUEST e COLON e
                e ASOP e
                e ASSIGN e
                term

term:           term INCOP
                MUL term
                AND term
                MINUS term
                UNOP term
                INCOP term
                SIZEOF term
                LP type_name RP term %prec STROP
                SIZEOF LP type_name RP %prec SIZEOF
                term LB e RB
                term LP RP
                term LP elist RP
                term STROP NAME
                NAME
                ICON
                FCON
                STRING
                LP e RP

type_name:      type abst_decl

abst_decl:      /* empty */
                LP RP
                LP abst_decl RP LP RP
                MUL abst_decl
                abst_decl LB RB
                abst_decl LB con_e RB
                LP abst_decl RP
```